VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Rozšiřování simulací v nástroji Kaira

Extending Simulations in The Tool Kaira

2015 Martin Palásek

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Martin Palásek**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Rozšiřování simulací v nástroji Kaira
Extending Simulations in The Tool Kaira

Zásady pro vypracování:

Kaira je nástroj určený pro modelování a generování paralelních/distribuovaných aplikací. Aktuálně jsou generovány C++ aplikace využívající MPI. Hlavním cílem práce bude implementovat lepší podporu simulací v tomto nástroji. Cíle práce lze shrnout v těchto bodech.
1. Seznamte se s nástrojem Kaira a aktuálním řešením simulací v tomto nástroji.
2. Navrhněte množinu rozšíření, které zjednoduší použití simulací. Například různé možnosti automatického provádění nebo automatické zastavení simulace.
3. Navržená rozšíření prakticky realizujte.
4. Demonstrujte nové možnosti simulací na příkladech.
K řešení využijte aktuálně používané technologie, specificky jazyky Python a C++.

Seznam doporučené odborné literatury:

[1] S. Böhm, M. Běhálek: Usage of Petri nets for high performance computing, Functional high-performance computing, ser. FHPC '12. New York, NY, USA: ACM, 2012, pp. 37–48.
[2] Domovské stránky projektu Kaira - http://verif.cs.vsb.cz/kaira/

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Ondřej Meca**

Datum zadání: 01.09.2013
Datum odevzdání: 07.05.2015

VYSOKÁ ŠKOLA BÁŇSKÁ - TECHNICKÁ UNIVERZITA OSTRAVA * Fakulta elektrotechniky a informatiky *

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou/diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 6. Května 2015 .....

Rád bych poděkoval mému vedoucímu Ing. Ondřeji Mecovi za vedení a konzultace při vytváření této práce. A také ostatním členům týmu, který vyvíjí nástroj Kaira, za rady, které dopomohly k dokončení této práce.

Abstrakt

Tato bakalářská práce se zabývá rozšířením nástroje Kaira. Nástroj Kaira slouží pro tvorbu MPI aplikací. Programování v nástroji Kaira je založeno na tvorbě barevných Petriho sítí. Hlavním tématem této práce jsou Simulace. Simulace jsou část nástroje Kaira, kde simulujeme programy Kairy. Hlavní náplní je zachytit jednotlivé kroky neboli stavy simulace. Umožnit uživateli návrat do předchozích stavů. Synchronizovat komunikaci uživatelského rozhraní simulace s vygenerovanou aplikací. A vytvořit vhodné úložiště pro jednotlivé stavy. Dalším tématem je vytvořit automatickou simulaci. Tato automatická simulace bude programy simulovat bez zásahu uživatele.

Klíčová slova: nástroj Kaira, Message Passing Interface, barevné Petriho sítě, simulace, aplikace, uživatelské rozhraní, automatická simulace

Abstract

This bachelor thesis deals with extensit of Kaira tool. Kaira tool is designed for creating MPI applications. Programming by Kaira tool is based on creating Coloured Petri Nets. Head topic of this thesis is simulations. Simulations are parts of Kaira tool, where we simulate Kaira programs. The main content is save single actions or states of simulation. Allow the user to return to previous states. Synchronize communication user interface of simulation with generated application. And create appropriately storage for single states. Next topic is create automatically simulation. This automatically simulation will simulate without user intervention.

Keywords: Kaira tool, Message Passing Interface, coloured Petri Nets, simulation, application, user interface, automatically simulation

Seznam použitých zkratek a symbolů

MPI	–	Message Passing Interface
GPL	–	General Public License
ID	–	Identifikační číslo
HTML	–	Hyper Text Markup Language
C++	–	C plus plus, programovací jazyk
Python	–	Programovací jazyk
FIFO	–	First In First Out
PTP	–	Project-To-Program
Gui	–	Graphical User Interface
Libs	–	Libraries
TreeView	–	Zobrazuje informace v hierarchické struktuře pomocí sbalitelných uzlů

Obsah

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Cílem této práce je rozšíření volně šiřitelného nástroje Kaira, který slouží pro tvorbu MPI aplikací. Blíže pak rozšířením částí Kairy, kterou nazýváme simulace. V těchto simulacích se budeme zabývat jednotlivými stavy, do kterých se simulace dostávají. Budeme řešit jejich ukládání a navracení do starších stavů.

Nejprve bude nutné seznámit se s nástrojem Kaira. Základními funkčními principy se budeme zabývat v kapitole 2. Popíšeme si zde jak Kaira funguje, řekneme si něco o Petriho sítích na kterých je Kaira založena.

V další kapitole 3 pojmenované Simulace se budeme zabývat stejnojmennou částí Kairy. Povíme si k čemu simulace slouží a jak fungují. V závěru se budeme zabývat tím, jak simulace vypadaly na začátku, před moji úpravou.

V kapitole 4 s názvem Rozšíření simulací se budeme zabývat tím, jakým způsobem byly simulace rozšířeny. V této kapitole se na rozšíření zaměříme z pohledu rozšíření knihoven psaných v jazyce C++. Zajímat nás budou hlavně jednotlivé stavy, do kterých se simulace dostává a uložení těchto stavů.

V následující kapitole 5 s názvem Rozšíření uživatelského rozhraní budeme na ukládání stavů pohlížet ze strany uživatelského rozhraní. Budeme se zabývat ukládáním pouhých obrazů stavů, také sladěním komunikace s aplikací, bude řeč o návratu do starších stavů a rozšíření historie těchto stavů.

Další kapitola 6 budeme řešit automatickou simulaci Kairy, a její ovládaní (spuštění, zastavení). Budeme rozebírat jakým způsobem musíme vybírat další kroky simulace, tak aby proběhla celá.

Na závěr se v kapitole 7 podíváme na další možná řešení toho rozšíření. Probereme výhody a nevýhody alternativních řešení, ale i mého řešení.

2 Kaira

Kaira je volně šiřitelný nástroj pod licencí GPL. Slouží pro vývoj a tvorbu MPI aplikací. Nabízí nám jednotné rostředí pro programování, testování, ladění, profilování a ověřování těchto aplikací [7]. Výsledný program v Kaiře vzniká kreslením sítě. Tato síť vychází z barevných Petriho sítí. Pomocí těchto sítí popíšeme paralelní část programu. Do této sítě pak můžeme vkládat sekvenční části programu. Tyto sekvenční části jsou tvořeny C++ zdrojovým kódem.

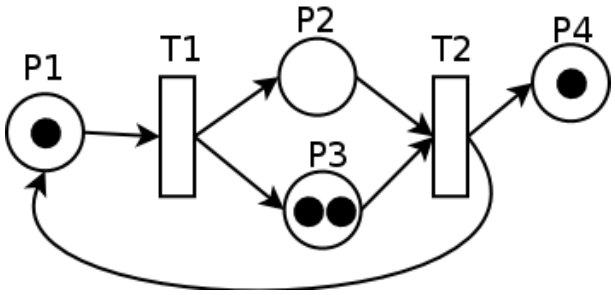
2.1 Petriho sítě

Petriho síť [2] je univerzální grafický a matematický nástroj pro návrh, modelování a formální analýzu dynamických systémů. Podle definice můžeme říct, že je Petriho síť reprezentována bipartitním orientovaným multigrafem. Zjednodušeně řečeno je Petriho síť soustavou prvků, které mají předem definované chování a pravidla. Na Petriho síti nalezneme tři základní prvky. Prvním prvkem jsou místa (places), nalezneme je na síti vyobrazeny jako kružnice. Místa na Petriho síti slouží k uchování dat, data jsou na síti reprezentovány v podobě tokenů. Druhým prvkem Petriho sítě jsou přechody (transitions), jsou vyobrazeny jako čtverce nebo obdélníky. Pokud daný token projde přes přechod, změní se stav sítě. Přechody tedy udávají chování sítě. Třetím prvkem Petriho sítě jsou hrany (arcs), na síti jsou reprezentovány šipkami. Hrany vytváří logické vazby na síti. Spojují vždy místo a přechod, nebo přechod a místo. Hrany nesmějí spojovat místo a místo, nebo přechod a přechod. Výsledná síť je pak dána místy, které jsou pomocí hran propojeny s přechody a opět pomocí hran zpět do míst. Stav sítě udávají data (tokeny), které se v danou chvíli nachází v místech, pokud jeden token projde přes přechod do nového místa, změní se stav celé sítě. Na obrázku 1 můžeme vidět C/E Petriho síť.

2.1.1 Typy Petriho sítí

Petriho sítí existuje několik odlišných variant. Všechny varianty vycházejí ze stejné myšlenky, ale snaží se nějakým způsobem upravit modelovací schopnosti sítě. Je pak na uživateli a na typu řešení aby si vybral správný typ sítě, který mu a jeho řešení bude nejvíce vyhovovat.

- *C/E Petriho sítě (Condition/Event Petri Nets)* je nejslabší z hlediska síly vyjadřování. Síť je určena následujcími prvky. Podmínkami v podobě kroužků. Událostmi v podobě obdélníků (nebo úseček). Šipkami vedoucími od podmínek k událostem nebo od událostí k podmínkám. A Tokeny, které jsou zobrazeny jako tečky v kroužcích (podmínkách). Každá podmínka pak může obsahovat vždy jeden token.
- *P/T Petriho sítě (Place/Transition Petri Nets)* jsou sítě, které mají v každém místě buď žádný, nebo jeden a více tokenů. Tímto počtem tokenů je určen stav sítě.
- *Petriho sítě s prioritami (Petri Nets with Priorites)* vznikly rozšířením P/T Petriho sítí. Tyto sítě mají navíc přidávají ke každému přechodu nezáporné celé číslo, které udává prioritu přechodu.



Obrázek 1: Petriho síť

- *Časové Petriho sítě (Timed Petri Nets)* opět vycházejí z P/T Petriho sítí a obohacují síť o časový pojem. V síti pak můžeme modelovat děje, které spotřebovávají různou časovou hodnotu. Děje můžeme pak rozdělit na deterministické, to jsou děje s konstantní dobou trvání, dále na děje stochastické, které mají náhodnou dobu trvání. Můžeme využít dějů kombinovaných, které kombinují deterministické a stochastické doby trvání.
- *Barevné Petriho sítě (Coloured Petri Nets)* [3] jsou dalším typem sítí založených na P/T Petriho sítích. V tomto typu sítí mohou být různé typy tokenů. Tyto tokeny nabývají různých hodnot(barev). Pro zpracování různobarevných tokenů je tento typ Petriho sítí rozšířen o prvky jako jsou proměnné, deklarování typů, inskripční výrazy hran, podmínky proveditelnosti (guards) a akce přechodů. Místa v této síti mohou obsahovat multimnožiny tokenů, přechody mohou obsahovat podmínky pro provedení přechodu.
- *Hierarchické Petriho sítě (Hierarchical Petri Nets)* jsou jako předešlé tři typy založeny na P/T Petriho sítích. Tyto sítě zavádí možnost hierarchického strukturování.
- *Objektové Petriho sítě (Object-Oriented Petri Nets)* jako předchozí typ jsou založeny na P/T Petriho sítích. Jak název napovídá jde o rozšíření Petriho sítí o koncept objektové orientace.

2.1.2 Výhody Petriho sítí

Díky grafickému znázornění Petriho sítě, může být Petriho síť srozumitelná i pro osoby, které nejsou detailně seznámeny s kompletní teorií Petriho sítí. Chování každé sítě je popsáno její sémantikou. Například o C/E Petriho sítích můžeme říct, že dané místo v případě C/E Petriho síti ho nazýváme podmínkou (condition) může uchovávat právě jeden token. Přechod je v C/E síti definován jako událost (event) a uskutečněním této události dojde ke změně stavu sítě. Pokud od podmínky vede šipka k události, nazýváme tuto podmínku jako vstupní podmínku (precondition). A jestliže vede šipka od události k podmínce je tato podmínka výstupní podmínkou (postcondition). Událost je poté proveditelná, pokud všechny její vstupní podmínky splněny a zároveň všechny její výstupní podmínky jsou nesplněny. Po tom co se provede tato proveditelná událost, se

změní stav sítě. Všechny vstupní podmínky jsou po té nesplněny a všechny výstupní podmínky jsou pak splněny. Následně můžeme říct, že událost byla provedena (fired). Pokud uživatel zná tyto základní údaje, měl by být schopný pracovat s C/E Petriho sítí aniž by znal kompletní teroii.

Další výhody Petriho sítí vycházejí z výše uvedených typů sítí. To že máme vyšší modelovací typ, ale nemusí znamenat, že bude zvolený typ lepším, než nižší modelovací typ. Vše se odvíjí od náročnosti řešení a tak se může stát, že nižší modelovací schopnosti sítě bude pro dané řešení lepší, než složitější typ sítě. Složité a rozsáhlé modely Petriho sítí můžeme hierarchicky popsat pomocí jednodušších a menších modelů sítí. Pokud potřebujeme provést v Petriho síti menší změny, nemusíme nutně měnit kompletní strukturu sítě.

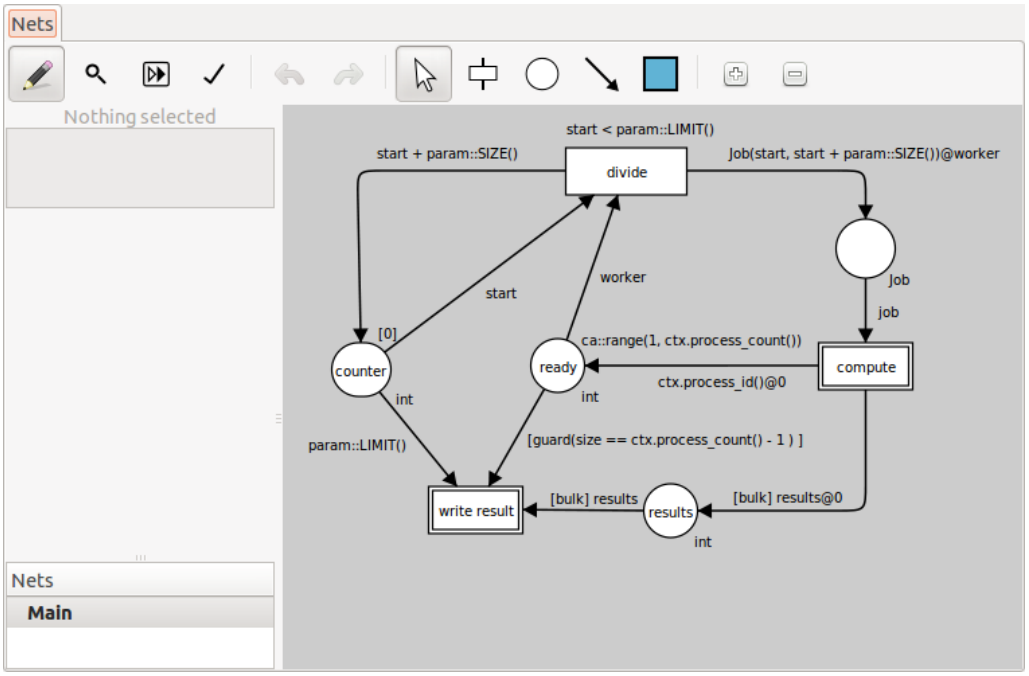
2.2 MPI

MPI [4] je knihovna, která se využívá pro realizaci masivně paralelních výpočtů za pomocí paralelního stroje s distribuovanou pamětí. MPI má interface pro programovací jazyky C a Fortran. MPI funguje ve zkratce tak, že existuje jedna spustitelná aplikace. Tato aplikace se spouští na předem zvoleném počtu procesů a každý tento proces běží odděleně na svém procesoru. Z toho vyplývá, že na každém procesoru je spuštěn ten stejný program a jednotlivé procesy si podle svého ID zjistí, kterou část programu mají vykonat. Jednotlivé procesy mezi sebou mohou komunikovat buď asynchronně, nebo synchronně, záleží na potřebě programátora. Procesy mezi sebou komunikují za pomocí přímého adresování přes ID procesu. Kromě operací pro jednotlivé procesy, mohou existovat globální operace pro všechny procesy, které jsou zaobaleny jedním komunikačním prostředím (Comm world).

2.3 Programování v Kaiře

Tvorba programu v Kaiře [1] se dá rozdělit do dvou částí. Prvním typem je vizuální programování, což je vlastně tvorba Petriho sítě. Druhým je vkládání sekvenčních zdrojových kódů do určitých částí sítě. Program v Kaiře může být tvořen pouze sítí, nebo sítí a sekvenčním kódem. Nikoliv však pouze sekvenčním kódem.

Tak jako Petriho síť obsahuje síť Kairy místa (places), přechody (transitions) a hrany (arcs). Navíc ještě může obsahovat inicializační zóny (init areas), které mohou nadefinovat více míst najednou. Navíc síť Kairy musí obsahovat spoustu dalších informací, tak aby bylo ze sítě možno vytvořit paralelní program. Například místa musí obsahovat datový typ. Hrany mohou být popsány výrazem, který určuje, za jakých podmínek bude token po hraně přenesen. Přechody a místa na síti v Kaiře pak mohou obsahovat sekvenční zdrojový kód. Na obrázku 2 můžeme vidět jak vypadá program workers ve vývojovém prostředí Kairy.



Obrázek 2: Program workers

2.3.1 Místa

Místa (places) jsou v programu zobrazena jako kružnice. Místa slouží jako paměť programu, fungují jako fronty tokenů. Fronta je typ paměti typu FIFO (First In First Out) to znamená, že token který přijde do místa jako první, také jako první odejde. Všechny místa v Kaiře mají tři základní vlastnosti: Název (Name), Typ (Type), inicializační výzaz (Init expression) a navíc mohou obsahovat inicializační kód (Init code).

- *Název (Name)* je řetězec, který popisuje název místa, vzhledem k programu nemá žádný význam. Na síti se zobrazí uprostřed kružnice, která reprezentuje dané místo. Pokud název nevyplníme odvodí se jeho název od identifikačního čísla, ale v tomto případě se název nezobrazí na síti.
- *Typ (Type)* může být jakýkoliv typ z C++. Na síti je typ zobrazen v v pravé dolní části místa.
- *Inicializační výraz (Init expression)* se používá k inicializaci tokenů v daném místě. Místo můžeme inicializovat několika způsoby. Pro příklad si uveďme naplnění místa čísly od nuly po číslo pět [0;1;2;3;4;5]. Nebo naplnění místa řetězci [”hello”,”world”].

2.3.2 Přechody

Přechody (transitions) jsou na síti zobrazeny v podobě obdélníků, definují chování programu. Přechody jsou na síti propojeny hranami, buď vstupními hranami, nebo výstup-

ními hranami. Do každého přechodu může vstupovat i vystupovat více hran. Přechody mají tři vlastnosti: Název (Name), Podmínka proveditelnosti (Guard) a Prioritu (Priority). Navíc mohou obsahovat zdrojový kód (Fire-Code), které bude proveden, vždy pokud bude token procházet přechodem.

- Název (Name) je stejně jako u místa řetězec, který jen reprezentuje název přechodu, a nemá žádny vliv na chování programu, název se zobrazí uprostřed přechodu. Stejně jako u místa pokud název neuvedeme je jeho názvem identifikační číslo, které se odvodí podle počtu všech prvků na síti.
- Podmínka proveditelnosti (Guard) je booleanovský C++ výraz, který může obsahovat proměnné, které se vyskytují na vstupní hraně. Slouží jako podmínka proveditelnosti přechodu. Můžeme v ní nadefinovat libovolnou podmínku. Pokud je tato podmínka splněna je přechod povolen, pokud není splněna přechod povolen nebude.
- Priorita (Priority) je hodnota která udává danému přechodu jeho prioritu. Pokud bude přechod povolen a bude mít prioritu vyšší než všechny ostatní přechody na síti, budou tyto přechody s nižší prioritou nepovoleny. Priorita se na síti zobrazuje jako malé číslo v pravé části přechodu.

2.3.3 Hrany

Hrany (Arcs) jsou na síti reprezentovány šipkami. V Kaiře máme dva typy hran, prvním typem jsou hrany vstupní (Input Arcs), a druhým typem jsou hrany výstupní (Output Ars). Oba typy mají odlišnou sémantiku. Všechny hrany na síti mají popis (Inscription) ve tvaru :

[configuration] main-expression@target

Kde konfigurace (configuration) se skládá z konfiguračních položek oddělených čarami, pokud chceme můžeme tuto část popisu hrany vynechat. Další je hlavní výraz (main-expression) je C++ výraz a definuje hodnotu tokenu která je buď převzata nebo umístěna do místa. A poslední částí popisu hrany je Cíl (Target) je C++ výraz, který definuje proces na který bude token poslán, tato část popisu hrany může být použita pouze na výstupní hraně. Do popisu vstupních hran můžeme do bloku konfigurace vpisovat tyto příkazy : bulk, filter(x), if(x), guard(x), svar(x), from(x), sort_by_source. Na výstupní hrany do popisu můžeme vepsat tyto příkazy: bulk, multicast, if(x), seq(x).

2.3.4 Sekvenční kódy

Sekvenční kódy (Sequential codes) umožňují do prvků v sítí umístit C++ zdrojový kód. Různé výrazy z jazyka C++ mohou být použity na různých prvcích Kairy, ale přechod umožňuje aby byl C++ kód umístěn přímo do něj. Tato část kódu se vykoná vždy při

provádění přechodu. Zdrojový kód přechodu je rozdělen na dvě části. V první je nadefinována struktura (Struct) s názvem Vars, tedy proměnné, ty se nadefinují automaticky podle vstupních hran a jejich popisů. Druhá část zdrojového kódu je funkce ”transition_fn” zde si můžeme napsat vlastní C++ kód. Přechod, který obsahuje zdrojový kód se poté na síti změní z obdélníku na dvojitě orámovaný obdélník. A pokud do je přechod spuštěn provede se tento sekvenční kód.

3 Simulace

V této kapitole si řekneme něco o simulacích v Kaiře. Nejprve se podíváme obecně, k čemu simulace slouží, jak se s nimi pracujeme a z jakých částí se skládá. Poté se podíváme, jak simulace fungují, jakým způsobem komunikují s aplikací. Hlavním cílem této práce je rozšířit právě tyto simulace, a tak si v závěru popíšeme a stanovíme původní stav simulací, ze kterého tato práce vychází.

3.1 Funkčnost simulací

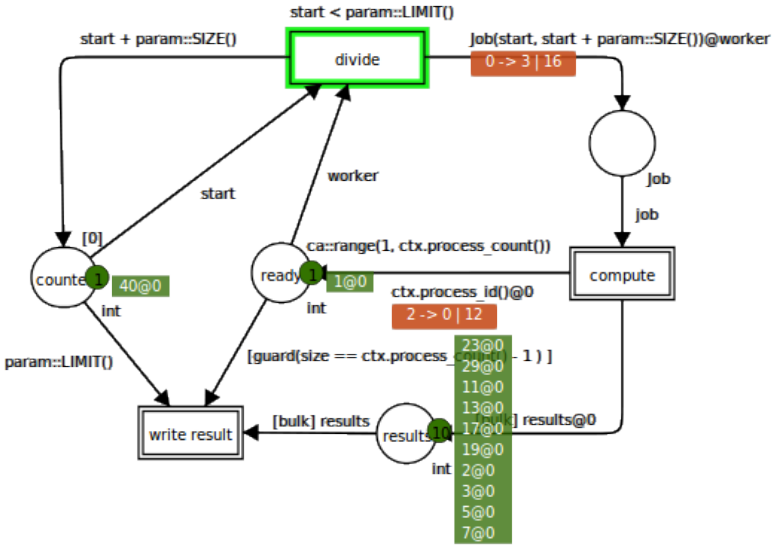
Simulace jsou částí Kairy, kde může uživatel interaktivně procházet programy vytvořené v Kaiře. Již víme, že program v Kaiře je založen na barevné Petriho síti. Z této sítě následně vychází i simulace. Při spuštění simulace se zobrazí vstupní okno, kde uživatel navolí počet procesů a vstupní parametry. Poté se vygeneruje a zobrazí nová forma sítě, už samotná simulace. Zde jsou zeleně ohraničeny proveditelné přechody. Dále jsou zde zobrazeny tokeny v podobě malých zelených rámečků s bílým textem a nachází se vždy u příslušného místa. Dále máme možnost zobrazit aktuální packety na síti, tato funkce je základně vypnutá a packety se přijímají automaticky. Tyto packety se nacházejí na přechodech, tedy mezi dvěma místy a tvoří tak jakési mezi stavy sítě. Pokud je funkce automatického přijímání vypnutá, packety se na síti zobrazí v podobě oranžových rámečků. Zelené tedy proveditelné přechody můžeme spouštět, oranžové packety můžeme přijímat a tak interaktivně měnit stavy sítě. Další funkčností je pak vypnutí automatického dokončování přechodu. Pokud tuto funkčnost vypneme, a spustíme proveditelný přechod, dostane se přechod do stavu spuštěn. Tento stav musíme ručně dokončit. Pod spuštěným přechodem se zobrazí žlutá šipka, pokud ji stiskneme, bude přechod dokončen. Na obrázku 3 můžeme vidět simulaci v Kaiře.

3.2 Přehled procesů (View)

V simulacích máme možnost volby mezi dvěma hlavními podfunkcemi. A to je Přehled procesů (View) a Historie simulací (History). Přehled procesů je náhled na jednotlivé procesy. V tabulce máme pod sebou seznam jednotlivých procesů, který vždy začíná políčkem "All", kde máme náhled na všechny procesy současně. Jednotlivé procesy se vždy číslují od 0, z toho plyne, pokud jsme si při spouštění simulace zvolili 4 procesy, poslední proces je uvedený jako proces 3. Pokud projíždíme jednotlivé procesy, na síti jsou povoleny jen procesy, které jsou povoleny v daném procesu. To samé platí u tokenů, na síti se zobrazí jen tokeny v daném procesu.

3.3 Historie simulací (History)

Další možností a také výhodou, kterou nám simulace nabízejí je historie neboli sekvence (sequence). Můžeme vidět na obrázku 6 a). Historie je jakýsi seznam všech akcí co byli na síti provedeny. Po zvolení položky v seznamu se nám vyobrazí stav sítě zachycený v daném momentu. Tento stav je pouze jakýsi obraz stavu, který si uživatelské rozhraní

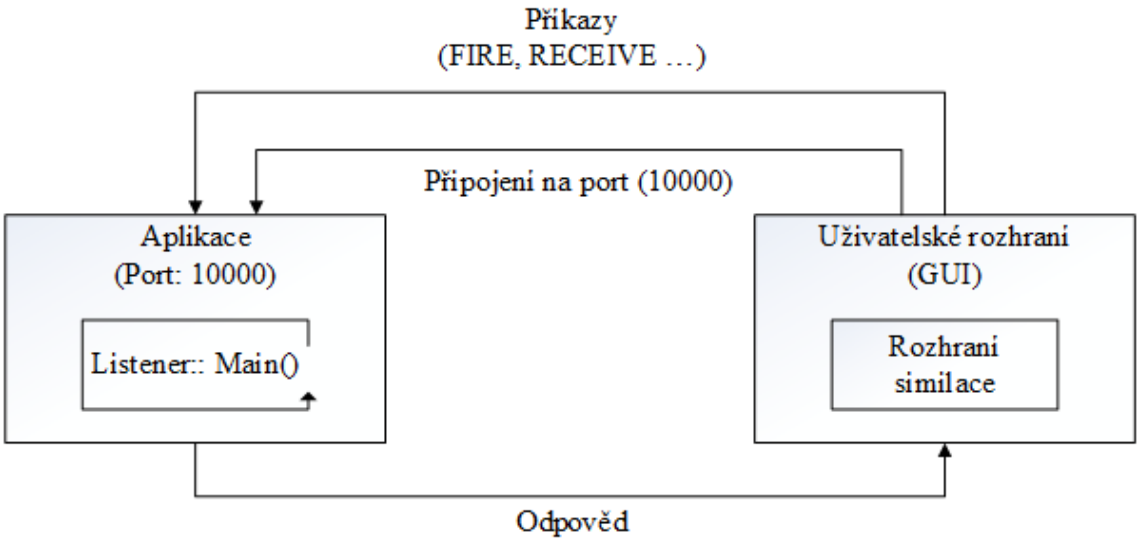


Obrázek 3: Simulace workers

zapamatovalo. To znamená, že není možné se k němu navrátit a rozvíjet, třeba jinou cestou. Pokud zkusíme spustit přechod ze stavu, který není aktuální zobrazí se chybová hláška (”A history of simulation is displayed, it cannot be changed”). Pokračovat tedy lze pouze v posledním stavu, který je zároveň aktuálním stavem sítě. Zpět k němu se dostaneme buď stiskem posledního stavu v historii nebo stiskem tlačítka pro zobrazení aktuálního stavu (”Show Current”).

3.4 Komunikace s aplikací

Pokud spustíme simulaci, nejprve se spustí aplikace. Tato aplikace je vygenerována při překladu námi nakreslené sítě v editoru Kairy. Při spouštění se přidávají doplňující informace (parametry) jako počet procesů, vstupní parametry sítě (ty si uživatel při spouštění simulace volí za pomoci uživatelského rozhraní). Následně se nastaví port, na kterém aplikace poběží (základně 10000). Posledním parametrem je parametr ”-b”, ten zajistí, že bude aplikace čekat na prvního uživatele, který se na ní připojí. V tomto případě se tuto aplikaci připojí uživatelské rozhraní. Pokud bude aplikace vygenerována a následně spuštěna výše uvedeným způsobem, automaticky se k ní připojí knihovna Libs/Cailie. V této knihovně se po spuštění aplikace začne provádět metoda Main, ta běží v nekonečném cyklu a naslouchá příkazům na daném portu.



Obrázek 4: Komunikace uživatelského rozhraní s aplikací

Ukažme si jednoduchý příklad komunikace. Pokud uživatel provede přechod číslo 104 uživatelské rozhraní zašle zprávu ”FIRE 104 0 2”, což znamená, že se má provést přechod číslo 104, 0 pak znázorňuje vlákno (Thread), a třetí parametr číslo 2 reprezentuje typ provedení přechodu, kdy se přechod provede kompletně (tzv. ”FULL FIRE”). V zdrojovém kódu se za pomoci podmínek, testuje prefix zprávy zaslané uživatelským rozhraním (v našem příkladu to byl prefix ”FIRE”), příkaz se dostává dále, kde se odečítají jednotlivé parametry a dle nich se vykonávají příslušné metody. Po dokončení těchto metod, zašle zase aplikace zprávu o vykonání pro uživatelské rozhraní. Komunikace je graficky znázorněna na Obrázku 4.

3.5 Původní stav simulací

Původní historie stavů funguje jenom jako jakýsi seznam ”obrázku”, ve kterých jsou zachyceny předchozí stavy průběhu simulace. V programu, se uchovávala jediná aktivní instance třídy State, která reprezentuje aktuální stav sítě. Při změně stavu simulace původní stav zanikl a na jeho místo se uložil nový stav. Z původního stavu se vytvořila jen vizuální interpretace v uživatelském rozhraní, tedy jen jakýsi ”obraz” toho původního stavu, ten si pamatoval jen tokeny na síti a aktivní přechody, což jsou informace potřebné k vizuální rekonstrukci stavu. Ovšem už neexistovala skutečná instance toho stavu v aplikaci, v tomto případě nebylo možné se ke stavu simulace navrátit a třeba ho rozvíjet jiným způsobem.

V samotné simulaci, jsme měli možnost vidět v levé části panel, který obsahuje dvě záložky, a to přehled procesů (View) a historii (History). Historie (History) obsahovala seznam stavů, pomocí těchto stavů se dala historie jen procházet, a zobrazovat její vizu-

ální interpretaci. Pod tímto seznamem jsou dvě tlačítka, jedno na uložení sekvence (Save sequence) a tlačítko pro zobrazení aktuálního stavu (Show curent). Zpět k seznamu historie, tento seznam vzniká v třídě SequenceView, která je odvozená od třídy SimpleList. Ve třídě SequenceView jsou uloženy jednotlivé záznamy tohoto seznamu. Každý záznam obsahuje informaci o procesu a vláknu (P/T, tedy Process/Thread), dále má informaci o akci (Action), kterou daný prvek reprezentuje (fire, start, finish, receive) a nakonec má informaci Arg, kde je zapsán například přechod který byl proveden, nebo tato část zůstává prázdná. Prvky se do seznamu vkládají pomocí metod připravených zvlášť pro každou akci (fire, start, finish, receive). Tyto metody se postarají o uložení záznamu, k přímému vložení do datové struktury využijí metodu Append, kterou třída SequenceView zdědila od třídy SimpleList. Dále je nutné zmínit metodu Clear, která je součástí třídy SimpleList, tato metoda vymaže celý seznam a využívá se jí při inicializaci seznamu, kde nám zajistí to, že bude na začátku seznam prázdný.

4 Rozšíření knihoven simulací

Rozšíření simulací je hlavní náplní této bakalářské práce. Jak bylo uvedeno výše, simulace jsou ve stavu, kdy procházení jednotlivých kroků je možné jen vizuálně, ale nelze se vrátit do libovolně zvoleného stavu. A právě problémy jako jsou, možnost vrátit se do jednotlivých bodů historie, možnost procházet síť alternativními cestami, a také možnost vizuálně si porovnat jednotlivé cesty sítí, bylo nutné vyřešit.

Dříve než se začneme zabývat samotným rozšířením Kairy, bylo by dobré alespoň zjednodušeně říct jak vypadá architektura Kairy. Architektura obsahuje tři základní části, Gui (Graphical user interface, nebo-li uživatelské rozhraní), PTP (Project-To-Program, neboli překladač projektu na program) a Libs (Libraries, neboli knihovny). Knihovny (Libs) jsou rozděleny na Cailie, což je knihovna, která se vždy spojuje se samotným programem, a další čtyři, které se připojují podle typu sestavení (Build Type), a těmi jsou CaVerif, CaSimrun, CaOctave, CaClient. Uživatelské rozhraní pak není nijak logicky rozděleno do složek, ale vzhledem k rozšiřování simulací nejvíce využijeme soubory Simulation.py, Simview.py, Colntolseq.py.

4.1 Rozšíření simulací

Hlavním úkolem této práce je navrhnout a rozšířit simulaci takovým způsobem, aby bylo možné libovolně se vracet k jednotlivým stavům a následně vytvářet alternativní průchody sítí. Tato myšlenka plyne ze situace, kdy máme více povolených přechodů a je na uživateli, který přechod zvolí a provede ho. Ovšem pokud byli simulace ”lineární” měl uživatel v rámci jedné simulace možnost vytvořit jeden průchod sítí od začátku k cíli. Následně bylo možné si tuto historii průchodu (nebo-li sekvenci) uložit a simulaci opakovat s tak se pokusit vytvořit alternativní průchod sítí a následně tyto průchody složitě porovnávat.

Výše uvedený problém má několik způsobů řešení. Nejjednodušším řešením může být umožnit uživateli se jen navracet do zvoleného stavu, ale neuchovávat stavy předcházející. Toto řešení by mohlo užitečné v případě, že uživatel bude chtít vyzkoušet různé alternativní cesty sítí, ale nebude je chtít například porovnávat s předchozími průchody, jelikož tyto předchozí obrazy sítě nebudou již existovat. Druhým řešením pak může být umožnit uživateli při procházení sítě vytvářet stromovou strukturu. Tedy umožnit mu navracet se do libovolně zvoleného stavu a uchovat pro něj předchozí stavy. Uživatel tedy následně může tvořit alternativní průchody, navracet se k předchozím průchodům a podle toho jak bude potřebovat, může s uloženými stavy pracovat. Dalším návrhem řešení může být vytvořit z této stromové struktury graf. Uživatel by měl tu samou funkčnost, ovšem můžeme předpokládat, že dva různé průchody sítí se dostanou do stavu, který bude pro oba průchody stejný. Zjednodušeně řečeno tak jak by průchod sítí umožňoval rozdvojení průchodu, tak by umožňoval i spojení zpět do jednoho stavu.

Momentálně byla pro tuto práci zvolená prostřední varianta, tedy varianta s vytvořením stromové struktury, která umožní uživateli při práci se simulací, vytvářet alternativní průchody a porovnávat nové stavy sítě s předchozími, bez toho aby musel neustále spouštět nové simulace.

4.2 Ukládání stavu v Libs

Doposud jsme se bavili především o obrazech simulace. Skutečný stav reprezentuje instance třídy State, tato třída se nachází v souboru State.h ve složce libs a pod složce cacalie. Tyto stavy je tedy nutné nějakým způsobem uchovávat. Pro ušetření místa v paměti bude výhodnější, pokud budeme uchovávat jen reference (odkazy) na tyto instance uložené v paměti. Tyto reference budeme následně ukládat do nějaké datové struktury, kterou bude možno snadno procházet a navracet se tak k jednotlivým stavům.

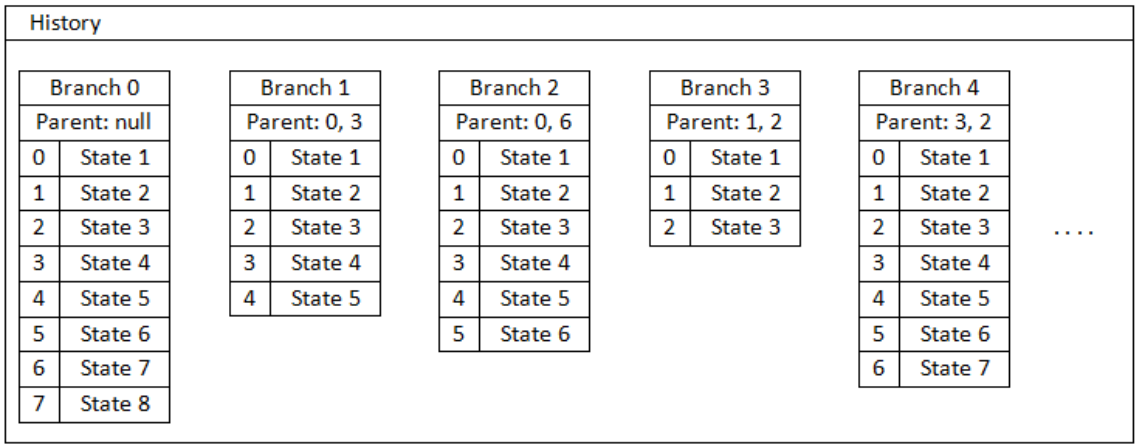
4.2.1 Datová struktura

Datovou strukturu bude reprezentovat jakýsi strom, zatím ale spíše budeme hovořit o datové struktuře nežli o stromu. Předtím, než datová struktura vznikla, bylo nutné se zamyslet, jak bude tato struktura využívaná. Jak často budeme ukládat stavy a jak často se ke stavům budeme navracet. Zda budeme pokaždé chtít rozvinout všechny cesty. Hlavní požadavky na strukturu, kterou budeme využívat jsou: jednoduché vkládání stavů a fakt, že uživatel většinou nebude rozvíjet všechny možné cesty, ale jen některé a to ve většině stavů od počátku cesty až do konce.

Teď když jsme si řekli požadavky na datovou strukturu, můžeme si ujasnit, jak naše struktura vypadá. Základním stavebním kamenem je Vektor (Vector) [5]. Vektor je už sám o sobě datovou strukturou, má spoustu vlastností, které nám usnadní práci. Největší výhodou je možnost přístupu k jednotlivým prvkům vektoru pomocí indexů a možnost vkládání prvku na konec vektoru. Pokud tedy vytvoříme vektor pro jednotlivé stavy, ty následně jeden po jednom budeme vkládat vždy nakonec vektoru, stačí nám následně znát pořadí (index) a dostaneme se vždy k námi požadovanému stavu.

Výše zmíněný vektor budeme nazývat History tedy jakási historie stavů. Tento vektor je součástí třídy Branch (větev). Třída Branch tedy obsahuje vektor History, kde jsou uloženy samotné stavy, tedy přesněji odkazy na ně. Dále obsahuje metody pro uložení stavu (save_state), pro nastavení stavu (set_state). Pro zjištění zda je vektor History prázdný (is_empty). Další metoda zjistí zda je aktuální stav posledním stavem vektoru (is_state_last), této metody se využívá pokud se nastavuje poslední stav z vektoru History. Další metody jsou pro nastavení, nebo zjištění aktuálního indexu a rodiče (tzv. "Settery a Gettery").

Zatím ale není struktura úplná, vytvořit totiž velký vektor History ve třídě Branch by bylo možná nějakým způsobem funkční řešení, ale odporovalo by to tomu, že chceme historii stavů rozvinout jako strom. Proto byl vytvořen ve třídě Listener, o které se ještě zmíníme, jiný vektor. Ten má název Branches (větve) a uchovává instance třídy Branch,



Obrázek 5: Datová struktura

nebo přesněji opět pouze odkazy (reference) na tyto instance. Tato třída opět obsahuje metody pro uložení stavu (save_state) a pro nastavení stavu (set_state), ovšem tyto metody už musí počítat s výběrem správné větve a indexu pro správné uložení stavu, a samotné uložení stavu pak provedou metody z třídy Branch. Na obrázku 5 můžeme vidět graficky znázorněnou tuto strukturu.

4.2.2 Ukládání stavů

Ve třídě Listener je metoda Main, která naslouchá příkazům z uživatelského rozhraní (Gui). Stav sítě se mění vždy, když je přechod na síti proveden (Fire transition), nebo pokud se příjme paket na síti (Receive packet). Toto přijetí paketu se provádí automaticky, pokud ovšem nezvolíme jinak viz. Kapitola 3. V praxi to pak funguje tak, že pokud dostaneme zprávu od uživatelského rozhraní o provedení přechodu, nebo přijetí paketu, je jisté, že se bude měnit stav sítě. Tudíž těsně před změnou stavu musíme aktuální stav uložit, a až poté z něj může vzniknout nový stav. Při ukládání stavů je nutno zjistit, zda se nejedná o uložení úplně prvního stavu, pokud ano je nutné vyhovořit první instanci třídy Branch, aby bylo možné první stav, ale i následující stavy ukládat. Při ukládání dalších stavů a vytváření nových větví, už není nutné vytvářet v této metodě instance třídy Branch, jelikož se o to stará metoda pro nastavení stavu (set_state).

Hodně důležité je nezapomenout uložit stav, při volání metody pro nastavení stavu (set_state). V momentě kdy tuto metodu voláme, nemáme totiž uložený poslední stav, jelikož stavy ukládáme těsně před provedením stavu (Fire transition) nebo přijetím paketu (Receive packet). Z toho plyne, že po nastavení stavu by zůstal vždy poslední stav dané větve neuložený a nešlo by se k němu vracet.

4.3 Nastavení stavu v Libs

Nastavení stavu probíhá tak, že si uživatel vybere stav v uživatelském rozhraní a stiskne tlačítko pro nastavení stavu. Z uživatelského rozhraní tedy přijde příkaz pro nastavení stavu "SET_STATE". Tento příkaz musí mít dva parametry, jinak bude automaticky zahozen. Celý příkaz tedy vypadá následovně "SET_STATE 0 3". Parametry příkazu jsou nutné, jelikož první parametr udává větev (branch) a druhý parametr udává pořadí (index). Pokud by nebyly zadány, vypíše se do konzole chybová hláška o špatných parametrech. Také není možné zavolat tento příkaz s parametry, které jsou například větší, než je počet větví, nebo počet stavů. Pokud se tak stane, opět se vypíše chybová hláška, která informuje o těchto chybách. Také není možné volat tento příkaz, pokud není žádný stav uložený.

Pokud tedy známe tyto dva parametry a jsou validní, můžeme je předat do metody pro nastavení stavu (set_state, Výpis zdrojového kódu 1) ve třídě Listener. Ta pomocí parametru, který udává větev (branch) najde příslušnou instanci třídy Branch a v ní následně podle indexu vyhledá příslušný stav. Tento stav, se znovu vytvoří jako nová instance a následně se nastaví jako aktuální stav sítě. Pokud nastavíme libovolný stav, automaticky se vytvoří nová větev (instance třídy Branch). Dalo by se říct, že tato větev vychází z tohoto stavu, abychom později věděli, kde větev vznikla, musíme zajistit nastavení souřadnic rodiče (branch a index). Teď můžeme vložit tuto větev (instanci třídy Branch) do vektoru větví (Branches), kde je připravena pro ukládání nových stavů.

```
void Listener::set_state(int branch, int index)
{
    Branch *CurrBranch = NULL;
    CurrBranch = Branches[branch];

    state = CurrBranch->set_state(index);
    State * NewState = new State(*state);
    state = NewState;

    Branch *NewBranch = NULL;
    NewBranch = new Branch();
    NewBranch->set_parent(branch, index);
    CurrentBranch = Branches.size();
    Branches.push_back(NewBranch);
}
```

Výpis 1: Nastavení stavu ve třídě Listener

Nově vzniklou větev je nutné si zapamatovat, tedy alespoň její pořadí neboli index ve vektoru větví (Branches). Všechny následující stavy sítě, které budeme ukládat, se budou ukládat právě do této větve, pokud uživatel znovu nenastaví stav a tím opět vytvoří novou větev. Pro zapamatování aktuální větve stačí jedna proměnná (v tom případě pojmenovaná Current_branch), do které si tuto pozici aktuální větve budeme ukládat.

5 Rozšíření uživatelského rozhraní

Předchozí kapitola se zabývala úpravou a rozšířením C++ části Kaira, řečeno jinak rozšířením knihoven Kairy. V této části se budeme zabývat úpravou a rozšířením uživatelského rozhraní, tak aby bylo možno využít nové struktury a funkce pro jednotlivé stavy. Uživatelské rozhraní bylo potřeba upravit tak, aby mohlo nejen využít všechny nové funkce, a také aby komunikace fungovala bez chyb a různých výjimek.

5.1 Ukládání stavů

I když ukládání opravdových stavů jak bylo zmíněno v minulých kapitolách je záležitostí C++ části Kairy, je důležité uchovávat stále i obrazy jednotlivých stavů. Tyto obrazy jsou totiž reprezentací daného stavu a v simulaci představují vizuální interpretaci stavu sítě pro uživatele. Pro tyto obrazy bylo nutné vytvořit datovou strukturu a nejlépe stejnou, nebo hodně podobnou jako pro skutečné stavy v C++. Ovšem uživatelské rozhraní Kairy je napsáno v jazyce Python a tak bylo k tvorbě datové struktury využito seznamu (List) [8]. Seznam v Pythonu má spoustu výhod, které ve výsledku usnadnili práci s datovou strukturou pro obrazy stavů v uživatelském rozhraní.

Celá datová struktura je navenek téměř podobná té co ukládá skutečné stavy, ale ve výsledku s ní pracujeme trochu jinak. Tato struktura je založená na hlavním seznamu, dalo by se říct páteři. Tento hlavní seznam v sobě uchovává další seznamy. Hlavní seznam potom nazýváme historie větví (history_branches). Další seznamy, které jsou v této historii větví uloženy, jsou v podstatě jednotlivé větve a tento seznam nazýváme historie instancí (history_instances). Dalším důležitým prvkem je proměnná, která udržuje index aktuální větve (current_branch).

Samotná funkčnost je také trochu odlišná, při provádění jednotlivých akcí na síti (Fire, Receive) se obrazy postupně ukládají do seznamu historie instancí. Pokud se uživatel rozhodne navrátit se k libovolnému stavu, teprve se celý seznam historie instancí uloží do historie větví. Obsah původního seznamu historie instancí se vymaže. Historie instací zůstane prázdná do doby, než se provede další akce na síti (Fire, Receive), poté se opět začne plnit novými stavy.

5.2 Rozšíření sekvence (History)

V této kapitole si řekneme něco o úpravě a rozšíření historie stavů (History neboli sekvence). V tomto bloku simulací Kairy se zobrazují v seznamu záznamy jednotlivých stavů. Nutno říct, že pokud bude libovolný stav z Historie zvolen, stále uvidíme jen jeho obraz. Skutečný stav, který budeme moci dále rozvíjet, vznikne, až po stisknutí tlačítka "Set state", o kterém bude řeč v další části.

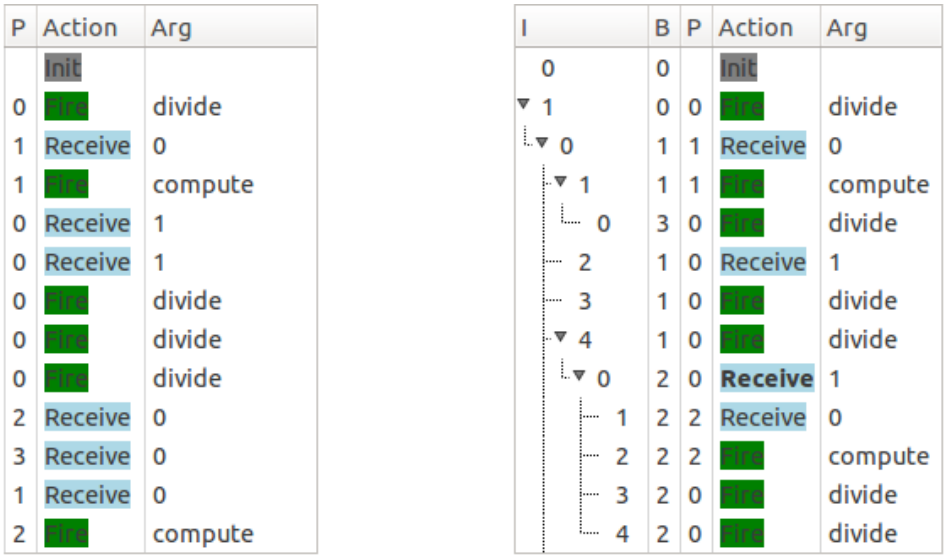
Původní verze historie stavů uchovávala o každém stavu tři informace. První informací je proces, o který se jedná. Druhou o jakou akci se jedná (Fire, Receive), prvky v

tomto sloupci mají podle své akce i barevné podbarvení. A třetí informací, o který přechod se jedná, zde je uveden název přechodu, pokud název nemá je uvedeno jen ID přechodu. Nově k původním přibyly další dvě informace. První informací je sloupec "I" neboli index. Index je informace o pocizi daného stavu ve větvi. Takže pokud má záznam v Historii ve sloupci index číslo 5, znamená to, že je to 6 stav ve větvi (indexujeme od 0). Druhým sloupcem, který přibyl je sloupec "B" neboli branch (větev). Branch určuje pozici dané větve v páteřním seznamu historie větví.

Nově už nemůžeme o historii stavů mluvit jako o seznamu. Jelikož byl seznam zaměněn za TreeView (stromový náhled) [6]. Toto TreeView dovede zobrazovat položky v tomto případě stavy ve stromové struktuře. Pokud je u daného stavu zobrazena šipka, znamená to, že daný stav je rozšířen novou větví. A pokud v této nové větvi existují nové stavy, budou tyto nové stavy propojeny se svým rodičem čárkovanou čárou. Pokud bude uživatel tvořit rozsáhlé historie, má například pro zpřehlednění možnost skrýt jednotlivé větve stiskem výše zmíněné šipky u rodiče.

S příchodem TreeView bylo nutné pozměnit systém adresování jednotlivých stavů v Historii. Dříve si uživatel zvolil stav a pomocí knihovní funkce se zjistila cesta, která byla jednorozměrná. Byl to tedy vlastně index. Tento index se použil v historii instancí a tak se vyvolal příslušný obraz stavu. Teď když je dalo by se říct dvourozměrná datová struktura, musíme adresu stavu zjistit jiným způsobem. K zjištění pozice stavu potřebujeme znát index stavu a větev (index a branch). Tyto informace má každý stav v Histroii uloženy ve sloupci "I" a "B" (index a branch). Uživatel si zvolí příslušný stav, pomocí knihovní funkce se zjistí jeho adresa v TreeView, díky které se dostaneme k datům jednotlivých řádků. Ze zvoleného řádku se vyberou právě informace o indexu a větvi. Jelikož jsou informace v řádku uloženy v podobě řetězců, musíme index a branch přetypovat na čísla, abychom s nimi mohli efektivně dále pracovat. Teď když známe adresu stavů, můžeme použít upravenou metody pro vyvolání obrazu stavu (set_runinstance_from_history). Této metodě předáme index a branch zvoleného stavu a ona na základě těchto dvou informací zobrazí příslušný obraz stavu.

Celá historie stavů zaměněná za TreeView funguje následovně. Pokud spustíme simulaci a procházíme síť, nepoznáme žádnou změnu. Změna nastává ve chvíli stisknutí tlačítka pro nastavení zvoleného stavu (Set state). Stav, který byl zvolen a byl nastaven, se označil tučným písmem. Stal se tedy aktuálním stavem. A všechny další akce, provedené na síti (Fire, Receive) budou vycházet z tohoto stavu. V historii stavů se pak tyto akce budou vkládat pod nastavený stav jako potomci. Navíc první stav nové větve bude nadřazený ostatním stavům v historii. Pro snadnější představení je chování vyobrazeno a obrázku 6 b). Toto chování TreeView je důsledkem toho, že pozice každého prvku v našem případě stavu TreeView, je určena rodičem (stavem ze kterého vychází). A poté je tato pozice určena pořadím, ve kterém byl stav vložen. Tedy pokud bude stav vložen do větve jako první, bude na prvním, druhý vložený bude na druhém místě, a tak dále. Proto bylo nutné první stav větve nadřadit ostatním. Pokud by tak nebylo a nastala by



Obrázek 6: Historie stavů (a - původní; b- Rozsířená)

situace, kdy by se uživatel navracel do jednoho stavu podruhé. Jednotlivé stavy obou nových větví by splynuli do jedné větve TreeView. Toto řešení by bylo nevyhovující, proto bylo nutné odlišit jednotlivé větve, tak aby byla každá větev viditelně oddělena.

Změnou historie stavů ze seznamu na TreeView vznikl nový problém. Pokud byla historie jako seznam, byla lineární a poslední stav byl vždy ten aktuální. Z čehož vycházeli původní metody pro zobrazení aktuálního stavu a pro testování, zda zobrazený stav je i aktuálním stavem. Nově se ale aktuální stav může pohybovat kdekoliv. Proto musely být metody pro zobrazení aktuálního stavu a pro testování, zda se jedná o aktuální stav přepracovány. Přepracovány tak aby počítali novou datovou strukturou a s faktem, že aktuální stav se nemusí nacházet vždy na konci. Aby uživatel poznal, který stav v jeho historii je aktuálním stavem, je nutné tento aktuální stav nějak odlišit. Proto vznikly nové metody, které pomocí adresy TreeView, nebo přímo odkazu na stav v TreeView odliší aktuální stav. Tyto metody buď přidáním HTML tagů zvýrazní stav historie, nebo odebráním těchto tagů zvýraznění odstraní. Můžeme vidět na obrázku 6 b), kde je stav s hodnotami 0 2 0 Receive 1 zvýrazněn.

Pokud vkládáme jednotlivé prvky do historie, můžeme počítat s následující úvahou. Vždy nově vložený prvek bude aktuálním stavem, a tudíž ho musíme zvýraznit. Předchozímu prvku tohoto stavu je pak nutné zvýraznění odebrat. Toto odebrání je založeno na metodě, která pracuje s odkazem na stav, kterému potřebujeme zvýraznění odebrat. To můžeme vidět ve čtvrté části zdrojového kódu 2 (Pod podmínkou setstate_phase je rovno hodnotě 0). Další situací je pokud budeme nastavovat stav vícekrát jako jednou

za sebou, bez toho aniž by proběhla jiná akce. V tomto případě stačí pouze podle odkazu na rodiče odstranit zvýraznění z předchozího rodiče a označit nového. Tyto akce vykonává první část zdrojového kódu 2 (Pod podmínkou setstate_phase je rovno hodnotě 3). Změna nastává, pokud provedeme nastavení stavu. Ke zvýraznění nového stavu využijeme adresy TreeView. Při vložení dalšího stavu se nový stav vloží jako potomek původního a pro odebrání zvýraznění se využije odkazu na rodiče. Poté se musí upravit cesta k novému stavu, tak abychom podle ní mohli zjistit odkaz na nový stav, který bude rodičem následujícího. Tyto kroky se dějí v druhé části zdrojového kódu 2 (Pod podmínkou setstate_phase je rovno hodnotě 2). Další části je vložení druhého prvku větve (ten vychází z prvního prvku). Je nutné odstranit zvýraznění u prvního prvku, pomocí odkazu. Poté upravit adresu odkazu, tak aby adresovala nově vložený prvek. Za pomocí této adresy zjistíme odkaz na tento prvek. Tyto kroky se dějí v třetí části zdrojového kódu 2 (Pod podmínkou setstate_phase je rovno hodnotě 1).

```
def unbold_prev_row(self):
    if self.setstate_phase is 3:
        self.unbold_row(self._parent)
        self.setstate_state = False
        self.setstate_phase = 2
    elif self.setstate_phase is 2:
        self.unbold_row(self._parent)
        self.path = self.get_path(self._parent)
        self.path = self.modify_path(self.path)
        self._parent = self.get_iter(self.path)
        self.setstate_state = False
        self.setstate_phase = 1
    elif self.setstate_phase is 1:
        self.unbold_row(self._parent)
        self.path = self.modify_path(self.path)
        if self.setstate_state is False:
            self.current_iter = self.get_iter(self.path)
        self.setstate_state = False
        self.setstate_phase = 0
    elif self.setstate_phase is 0:
        self.unbold_row(self.current_iter)
        self.current_iter = self.iter_next(self.current_iter)
        self.setstate_state = False
```

Výpis 2: Odstranění zvýraznění předchozího řádku a nastavení odkazů TreeView

5.3 Nastavení stavu

Základním stavebním kamenem nastavení stavu, o kterém byla řeč v předchozí kapitole je tlačítko pro nastavení stavu (set_state). Nachází se pod Historií stavů, a po zvolení stavu v Historii a stisknutí tohoto tlačítka se změní aktuální stav sítě. Ovšem celé to není tak jednoduché, za touto akcí se skrývá spousta menších funkcí, které dohromady nastaví nový stav sítě.

Celé nastavení stavu začíná metodou ”set_state” ve třídě SimView. Nejprve je nutné zjistit index a větev stavu, a poté odkaz na tento stav v Historii stavu, jelikož nastavovaný stav bude vlastně rodič nového stavu. Poté je nutné ještě zjistit adresu v TreeView aktuálního stavu a zavolat metodu pro odstranění tagů, který zvýrazňují aktuální stav. Následně už se zavolá metoda ”set_state” ve třídě Simulation, se třemi parametry (index, branch, parent).

Další kroky se dějí ve třídě simulation. Nejprve se k hodnotě ukazatele na aktuální větev (current_branch) přičte číslo jedna, to plyne z toho, že nastavením stavu vzniká nová větev a ta bude mít označení o jedno vyšší než větev poslední. Následně se zavolá metoda pro odstranění označení posledního aktivního stavu a metoda pro označení nového stavu, který byl zvolen pro návrat. Tyto metody se nacházejí ve třídě SequenceView a pracují podle potřeby s odkazem na stav v historii stavů, nebo s adresou stavu v historii stavů. Teď už se jen zavolá metoda, která vloží historii instací do historie větví a původní historii instancí smaže a tak jí přichystá pro novou větev. A nakonec se pošle požadavek ”SET_STATE” na aplikaci, který nastaví stav zvolený na začátku jako aktuální stav spuštěné aplikace.

6 Automatická simulace

Dalším úkolem této práce bylo vytvořit automatickou simulaci. Automatickou simulací máme na mysli cyklický program, který bude postupně provádět proveditelné přechody, přijímat pakety a dokončovat spuštěné přechody. Tyto akce se budou provádět tak dlouho, dokud bude možné nějakou akci provést, nebo dokud uživatel simulaci nezastaví.

6.1 Volba akce

V této náhodné simulaci bude mít největší hodnotu proveditelný přechod. Pokud se na síti objeví proveditelný přechod, bude mít přednost před všemi ostatními akcemi a provede se jako první. Ovšem může se stát, že budou na síti v jednu chvíli dva a více proveditelných přechodů. Pak metoda random_enabled_transition, kterou vidíme ve výpisu zdrojového kódu 3, vybere náhodný přechod a ten bude proveden jako první. Metoda získá tuple (ntici) se všemi ID proveditelných přechodů a přetypuje ho na seznam (list). Pomocí knihovní funkce, se vybere jedeno ID náhodného přechodu. Toto ID proveditelného přechodu se stane návratovou hodnotou funkce.

```
def random_enabled_transition(self):
    enabled = self.perspective.get_enabled_transitions()
    enabled_list = list(enabled)
    if enabled_list == []:
        return
    else:
        random_tran = random.sample(enabled_list, 1)
        return(random_tran[0])
```

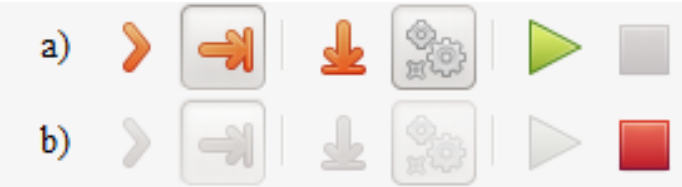
Výpis 3: Volba náhodného proveditelného přechodu

Druhou prioritu má dokončení přechodu. Pokud bude simulace nastavena pouze na spuštění přechodů, a nebude na sítí v danou chvíli žádný proveditelný přechod, budou se dokončovat spuštěné přechody. Výběr těchto spuštěných přechodů, už neprobíhá náhodně, ale první se dokončuje přechod na nejnižším procesu.

Třetí prioritou je přijímání paketů na síti. Za předpokladu pokud v danou chvíli není na síti žádný proveditelný přechod a žádný přechod není spuštěný. Nachází se na síti pouze pakety, bude právě paket s nejnižším ID procesu přijímán.

6.2 Tlačítka

V Kaiře také přibyli dvě nové tlačítka pro spuštění a zastavení simulace. Můžeme je vidět na obrázku 7. Kde zelené " Play " tlačítko simulaci spustí a červené " Stop " tlačítko simulaci zastaví. Pokud se simulace spustí, zamknou se všechny ostatní tlačítka pro konfiguraci simulace 7.b. Tak aby nebylo možné měnit konfiguraci za běhu simulace. Po



Obrázek 7: Tlačítka simulace

skončení simulace a nebo po ručním zastavení ” Stop ” tlačítkem se tyto tlačítka pro konfiguraci opět odemknou a je možné měnit nastavení simulace 7.a. Ve výpisu zdrojového kódu 4 můžeme vidět definici tlačítka pro spuštění simulace. Zde můžeme vidět, že po stisku tlačítka se zavolá metoda start_automatically_run.

```
button = gtk.ToolButton(None)
button. set_tooltip_text ("Automatically␣run")
button.set_stock_id(gtk.STOCK_MEDIA_PLAY)
button.connect("clicked", lambda w: self.start_automatically_run())
toolbar.add(button)
self.button_auto_run = button
```

Výpis 4: Definice tlačítka pro spuštění simulace

7 Další možná řešení

V této kapitole se budeme zabývat dalšími typy řešení hlavního úkolu této bakalářské práce. Jedná se o rozšíření historie stavů v simulacích. Mohou totiž existovat další typy řešení, které mohou být v dané situaci více či méně efektivní. Některé typy řešení, už byli naznačeny v kapitole 4.1. Vybrané z nich si v této kapitole projdeme detailněji. Nejprve se ale podíváme na nevýhody, které má mé řešení.

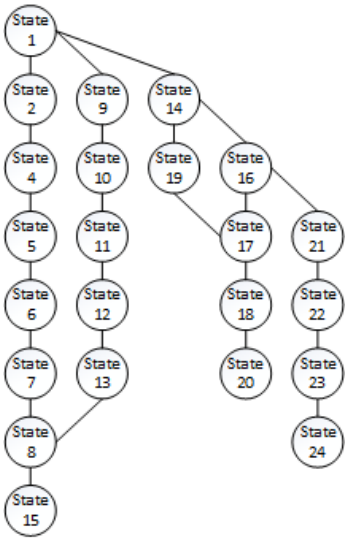
7.1 Nevýhody mého řešení

Dalo by se říci, že žádné řešení nebude ideální. Tak je to i v případě mého řešení. Hlavním problémem je paměť. Pokud bude uživatel procházet simulace klasických sítí, nejspíše se ho problém týkat nebude. Pokud bude chtít procházet nějakou větší síť a bude vytvářet velké množství stavů, bude kladen velký nárok na paměť. Jelikož jsou všechny stavy uchovávány právě v paměti. Dalším problémem může být množení stejných stavů. Pokud bude uživatel vytvářet alternativní průchody sítí, může nastat situace, kdy budou vznikat stejné stavy sítě. Tyto stavy, ačkoliv budou naprosto totožné, budou uloženy v paměti samostatně. Opět tedy vzniká velká náročnost na paměť programu.

Jak bylo řečeno, mé řešení je velmi náročné na paměť programu. Na druhou stranu je nutné říci, že při procházení menších sítí, bude zase mé řešení rychlejší a výhodnější. Jelikož máme všechny stavy k dispozici přímo v paměti a tak návrat k nim je v podstatě okamžitý.

7.2 Alternativní řešení: Graf

Prvním alternativním řešením může být zaměnění stromové struktury za grafovou. Vytvořit tedy jakýsi graf stavů. Kaira by pak sama sledovala jednotlivé stavy a testovala, zda se nový stav neshoduje se stavem, který již existuje. Pokud by se tak stalo, dvě nebo více větví stavů sítě by se opět sloučili do jedné a tím by vznikal graf stavů. Obrázek 8 U tohoto řešení by pak využívalo například hashování stavů. Každý stav by pak měl svůj hash. Následně by stačilo porovnávat jen tyto hashe k zjištění zda jde o stejný stav.
Toto řešení bude o trochu šetrnější k paměti, než mé řešení. Ovšem dá se předpokládat, že tvorba grafové struktury by nevyhovovala všem uživatelům. Toto řešení bude slučovat větve, vždy do stavu, který bude pro obě větve shodný. V určitých situacích by ale uživatel mohl požadovat, aby se stavy neslučovali a jednotlivé větve zůstaly odděleny.



Obrázek 8: Grafová datová struktura

7.3 Alternativní řešení: Rekonstrukce stavů

Další navrhnutý typ řešení je založen na úspoře paměti. Nejprve si ujasněme, že budeme vycházet ze stromové struktury. Z každé větve pak budeme do paměti ukládat pouze počáteční, tedy první stav větve. Pro každý následující stav si uložíme příkaz, který tento stav vytvořil (FIRE, RECEIVE atd.). Poté pokud bude uživatel požadovat návrat do příslušného stavu, vykonáme všechny dané příkazy, které tomuto stavu předcházeli. Tím získáme požadovaný stav sítě.

Toto řešení bude s největší pravděpodobností šetrné k paměti programu. Na druhou stranu pokud by uživatel tvořil dlouhé větve, návrat k posledním stavům větve vy mohl zabrat nějaký čas navíc. To znamená, že sice ušetříme na paměti, ale zase zatížíme operační paměť větším množstvím výpočtů.

8 Závěr

Cílem této bakalářské práce bylo rozšířit simulace v nástroji Kaira. Tyto simulace rozšířit takovým způsobem, aby bylo možné navracet se k jednotlivým stavům simulace. Nejprve ale bylo nutné pochopit, jak samotné Kaira funguje, seznámit se s problematikou Petriho sítí na kterých je tento nástroj založen. Tyto teoretické informace byly popsány v první kapitole. Následně bylo nutné začít se zabývat simulacemi Kairy. Pochopit jak fungují, jak komunikují s aplikací a zachytit jejich původní stav.

Další část se už zabývala praktickým řešením rozšíření simulací. Nejprve bylo nutné začít ukládat i starší stavy. Poté vytvořit strukturu pro reference (odkazy) na objekty stavů, tak aby k hledanému stavu byl snadný přístup. Bylo zvoleno řešení, kde každý stav má něco jako svou adresu. Tato adresa je určena číslem větve a indexem ve větvi. Každý stav je tak možné vyhledat pomocí těchto dvou údajů. Poté bylo nutné vytvořit metody pro rozpoznání zprávy od uživatele či uživatelského rozhraní. Ty jsou založeny na odečítání prefixu a parametrů. Pomocí těchto metod už bylo možné komunikovat s aplikací a tedy navracet se do předchozích stavů, i když jen v aplikaci. Proto, aby komunikace fungovala bezchybně, bylo potřebné sladit uživatelské rozhraní. Vytvořit v něm stejné úložiště pro obrazy stavů. Také vytvořit TreeView, které tvoří náhled na námi vytvořenou stromovou strukturu v paměti. A vytvořit ovládací prvky, tlačítka.

Druhým typem rozšíření simulací bylo vytvoření automatické simulace. Zde bylo potřebné vytvořit metody, které automaticky vybírají proveditelné přechody, spuštěné přechody a pakety na síti. Tyto metody následně pracují s původními metodami, které pracují přímo s aplikací. Tato automatická simulace také potřebovala ovládací prvky, nějaké tlačítka pro spouštění a zastavení této automatické simulace.

Tato práce řeší rozšíření historie stavů pouze jedním způsobem, další možné alternativní řešení byly rozebrány, v závěrečné kapitole. Zde jsme došli k závěru, že každé řešení má své výhody i nevýhody a tvrdit o jednom, že je efektivnější nelze. Ovšem přímo mé řešení by bylo možno rozšířit na grafovou strukturu, která je popsána v předchozí kapitole. Jelikož tato struktura je celkem podobná a ušetří místo v paměti. Automatickou simulaci by pak bylo možno rozšířit o různá nastavení, jako rychlost spouštění akcí nebo možnost volby, které akce budou prioritní.

V této práci jsem se poprvé setkal s oddělenou aplikací a uživatelským rozhraním, které aplikaci pouze zasílá požadavky. Sladění těchto dvou částí Kairy pro mě bylo prohloubení znalostí programovacích jazyků C++ a Python. Kde jsem se setkal s pro mne novými možnostmi tvorby aplikací a jejich uživatelských rozhraní.

Martin Palásek

9 Reference

[1] S. Böhm. *Unifying Framework For Development of Message-Passing Applications* [online]. VŠB-Technická univerzita Ostrava. Ostrava 2013 [cit. 2015-4-26]. Dostupné na: http://verif.cs.vsb.cz/sb/thesis.pdf

[2] J. Markl. *Petriho sítě* [online]. VŠB-Technická univerzita Ostrava. Ostrava 2006 [cit. 2015-4-29]. Dostupné na: http://www.cs.vsb.cz/markl/pn/index.html

[3] M. Češka, V. Marek, P. Novosad, T. Vojnar. *Petriho sítě* [online]. VUT v Brně. Brno 2009. [cit. 2015-4-29]. Dostupné na: http://www.fit.vutbr.cz/study/courses/PES/public/Pomucky/PES_opora.pdf

[4] I. Foster. *Designing and Building Parallel Programs* [online]. [cit. 2015-5-1]. Kap. 8. Dostupné na: http://www.mcs.anl.gov/~itf/dbpp/text/book.html

[5] *General information about the C++ programming language* [online]. [cit. 2015-5-3]. Dostupné na: http://www.cplusplus.com/

[6] *PyGTK: GTK+ for Python* [online]. [cit. 2015-5-3] Dostupné na: http://www.pygtk.org/

[7] *Kaira: High-level tool for MPI* [online]. [cit. 2015-5-3] Dostupné na: http://verif.cs.vsb.cz/kaira/

[8] *Python documentation* [online]. [cit. 2015-5-3] Dostupné na: https://www.python.org/doc/

A Obsah přiloženého CD

- Elektronicka verze této práce **pal0086_bp.pdf**
- Archív s Kairou, příkladem, textovým souborem Instalace a textovým souborem Editované_soubory **Archiv.zip**

A.1 Kaira

Upravená verze Kairy, s řešením této práce

A.2 Příklad

Soubor pdf, kde jsou na příkladu demonstrované nové možnosti Kairy

A.3 Instalace

V tomto textovém souboru jsou popsány jednotlivé kroky instalace Kairy

A.4 Editované soubory

V tomto textovém souboru jsou zapsány soubory, které jsem v Kaiře editoval, nebo do Kairy přidal při tvorbě této práce.